John T. Anderson

Engineering Note

Date: April 30, 2001
Rev Date: April 30, 2001

Project: 8-MCM AFE
Doc. No: A1010430

Subject: Generic notes on test software for AFE boards

Introduction

The purpose of this document is to provide a very basic outline of how the AFE Test Stand software is set up. It will no doubt expand as more development occurs.

General Philosophy

Test stand software is written as VBA code running underneath an Excel spreadsheet. All interface to the external world is through the Bit 3 card set to VME, from which access to MIL-STD 1553 and the SVX control cables is made. MIL-STD 1553 is used to control the device under test, plus the following devices:

- The AFE Power Supply Box, through a Rack Monitor
- An AFE Test Module which is connected to left-handed AFEs under test
- Another AFE Test Module, for testing right-handed AFE boards

The User Interface

The user interface is manifested as a *single* Excel worksheet, with the following rules:

- The user may not enter any data into the cells of the spreadsheet which may in any way modify the operation of the tests.
- All test execution is made possible only by the user hitting a button on the spreadsheet that calls a specific VBA subroutine which performs the test.
- A fixed area of the spreadsheet is reserved for output of prompts or other information to guide the user. All tests clear this area upon start and are individually responsible for controlling formatting to keep the information presented within bounds.
- Whenever a test requires input from the user (such as a 'continue' response, or a 'yes/no' or a value), the VBA code shall call a separate VBA subroutine which provides a consistent popup window to receive this kind of response. The user may only enter information through these popups, such that the display and format of the main spreadsheet is never violated. It is recommended that exactly three user input subroutines be implemented:
  - Getvald(), to return a long decimal value. (That is, the user types in decimal)
  - Getvalh() to return a long hexidecimal value. (That is, the user types in hex and the routine converts as required to return a long – probably this is best done by getting a string and converting from there).
  - Wait_for_continue(), which simply puts up a temporary button and waits for the user to press it.
- Whenever any test is run the color of the button pressed to run the test changes color to denote whether the test passed or failed. One button on the spreadsheet will have the function of resetting all buttons to 'grey' (test not yet run), and logging that that action has occurred. Each test button will then change color to either red (test failed) or green (test passed) upon completion of the test.
- Tests may be executed in any order desired by the user, and the responsibility of initialization rests with each individual test.

- One button on the spreadsheet will actuate a ‘startup’ test which does no testing of the hardware but simply obtains the identifying information about the test set (user name, date, which board, etc.).
- Hardware addresses of interfaces (whether VME or PC addresses), plus names of DLLs used and any other global information, shall be kept in a separate configuration file and shall NOT be presented on or be obtained from Excel.

Structure of Calls to the Hardware

Much of this already exists due to Bob Angstadt’s and Paul Rubinov’s diligent efforts. This section presumes that all the lower-level calls to the 1553 and SASEQ modules are present and only describes the very limited set of subroutines which are necessary for a developer to call to make one particular test of the AFE, without knowledge or interest in how the actual bits get passed through the various interfaces.

For documentation purposes, all calls are defined as

Function_name(arg1,arg2….) or retval = function_name(arg1, arg2,…)

Where the arguments to the function are given with a typing indentifier of *int, long, float* or *string*. *Int* will be interpreted as ‘sixteen bit integer’; *Long* means ’32 bit integer’ and *string* means ‘null-terminated string’. *Float* indicates a floating point number. Items are signed quantities unless preceded by the word *unsigned*. These definitions are meant to guide the software authors as to the expected numeric ranges of the arguments or function returns and may be implemented differently dependent upon the vagaries of Visual Basic for Applications.

AFE-Specific Calls

- Set_AFE_Address(int RT)

  An internal variable is set so that all further calls to AFE routines assume that the 1553 RT address is the value passed via *RT*.

- Write_AFE_Register(int address, unsigned int data)

  The sixteen bit value *data* is written to the dual-port RAM of the AFE, at location *address.* Values between zero and 0x7FF are legal for *address*, as there are only 2K locations in the dual-port RAM. The 1553 address of the AFE is taken from the global variable set by the last call to Set_AFE_Address().

- Unsigned int retval = Read_AFE_Register(int address)

  The sixteen-bit unsigned value from AFE dual-port ram location *address* is returned. Values between zero and 0x7FF are legal for *address*, as there are only 2K locations in the dual-port RAM. The 1553 address of the AFE is taken from the global variable set by the last call to Set_AFE_Address().

- Turn_SVX_ON()

  The AFE is told to execute command 0x08, which turns on all the SVX chips. It is presumed that the SASEQ is in the correct state to allow everything to work.

- Turn_SVX_OFF()

  The AFE is told to execute command 0x09, which turns all SVX chips off.

- Enable_SIFT_Clocks()

  The value 0x1F is written to DPRAM address 0x0302, then command 0x38 is issued to the AFE. This turns all SIFT clocks on in their normal operational mode.

- Disable_SIFT_Clocks()

  The value 0x00 is written to DPRAM address 0x0302, then command 0x38 is issued to the AFE. This turns all SIFT clocks off.

- Setup_VSVX(int enable_SIFT_data, int skip_other_chips, int pipeline_delay)

  The values from arguments 'enable_SIFT_data' and 'skip_other_chips' are combined together to form a word which is written to location 0x0180 of the DPRAM. The formula is

  *datavalue = 129 + ((enable_sift_chips and 0x01)*2) + ((skip_other_chips and 0x01) * 64).*

  As implied, the two arguments are boolean (legal values of 1 or 0 only). The pipeline_delay value, which is a number from 0 to 127, is written to location 0x0182 of the DPRAM. After loading these two locations, AFE command 0x0C is executed to load the controls into the VSVX.

- Unsigned long retval = Read_AFE_ADC(unsigned int channel)

  A generic A/D conversion is performed by the AFE and the sixteen bit value so generated is returned. The argument 'channel' is decoded by the underlying VBA code to select one particular readback of the AFE and AFE command 0x02 is used to perform the conversion.

- Set_Bias_voltage(int cable, float desired_voltage)

  The second argument is converted from a floating point number to a DAC setting using the conversion formula

  *Dacset = int(255 * (desired_voltage/10.0))*

  based upon the hardware implementation of an eight-bit DAC which generates 0 to 10 volts. This DAC setting is then loaded into the dac of interest. At present this is done via the generic DAC control command 0x17.

- Set_Heater_Drive(int cable, int drive)

  The second argument is converted from a floating point number to a DAC setting using the conversion formula

  *Dacset = int(255 * (desired_voltage/10.0))*

  based upon the hardware implementation of an eight-bit DAC which generates 0 to 10 volts. This DAC setting is then loaded into the dac of interest. At present this is done via the generic DAC control command 0x17.

- Enable_Cryo_loop()

  Writes a value to dpram which enables the cryo loop to run (some address in the 0x400 – 0x41F range, check with Stefan).

- Disable_Cryo_loop()

  Writes a value to dpram which disables the cryo loop, see Enable_Cryo_loop() above.

- Set_Desired_Temperature(int cable, float desired_temp)

  Writes the desired cryo temp setpoint for the given cable, addresses TBD. Talk to Stefan.

- Set_SIFT_Threshold(int MCM, float thresh_1, float thresh_2, float thresh_3, float thresh_4)

  Converts the four threshold arguments from *voltages* to *dac settings* using the formulae already present in my earlier threshold setting function. The routine then uses AFE command 0x05 to load the appropriate DACs based upon the 'MCM' argument.

- Set_SVX_VREFs(int MCM, float VREF_1, float VREF_2, float VREF_3, float VREF_4)

  Converts the four threshold arguments from *voltages* to *dac settings* using the formulae already present in my earlier VREF setting function. The routine then uses AFE command 0x05 to load the appropriate DACs based upon the 'MCM' argument.

SASEQ Functions

Same arguments as above, but for AFE testing, the list of necessary functions is smaller. Again, much underlying code exists thanks to the hard work of others.

- LOAD_AFE_SVX(int rhb_lhb, int nchips, int bandwidth, int pipe_depth, int read_all, int zero_suppress_thresh)

  Rhb_lhb is used to figure out which register of the SASEQ one writes to. Nchips, bandwidth, pipe_depth, read_all and zero_suppress_thresh are the only values of the SVX download one really wishes to manipulate. All other data can be fixed.

  Here's a first hack at what this might look like:

```
'   Subroutine to initialize SVX chips on an AFE.
'
'   Global constants which are expected to be known to the code:
'
'   SASEQ_BASE_ADDR

Sub LOAD_AFE_SVX(rhb_lhb As Integer, nchips As Integer, bandwidth As Integer, pipe_depth As Integer, read_all As
Integer, zero_suppress_thresh As Integer)
Dim chip_cnt, bit_cnt, scratch As Integer
Dim bitmask As Integer
Dim SASEQ_ADDR As Long

Const BITS_PER_SVX = 190

'
'   check for illicit values
'
rhb_lhb = rhb_lhb & 1
readall = readall & 1
'
'   Figure out vme address from global base and whether talking to lhb or rhb
'
SASEQ_ADDR = SASEQ_BASE_ADDR + (rhb_lhb * 2)

'
'   Initialize interface
'
scratch = VB_clrlatcherri   'clear Bit3 latched error bit

'
'   Start downloadin' bits
'
For chip_cnt = 0 To 7

    For bit_cnt = 0 To BITS_PER_SVX

    Select Case bit_cnt
        Case Is < 128       'test mask bits
            scratch = VB_Writew(SASEQ_ADDR, 0)
        Case 128            'test pulse polarity
            scratch = VB_Writew(SASEQ_ADDR, 1)
        Case 129            'pipeline voltage offset
            scratch = VB_Writew(SASEQ_ADDR, 0)
            bitmask = bandwidth 'prep for cases 130 thru 135
        '
        '   strip bits out for preamp bandwidth, which is loaded LSB
        '   to MSB.
        '
        Case 130, 131, 132, 133, 134, 135 'preamp bandwidth
            scratch = VB_Writew(SASEQ_ADDR, bitmask & 1)  'write lsb
            bitmask = bitmask / 2    'shift right to set up next bit
```

```
'
'   Chip ID.  Derive from counter to give IDs of A0,A1,A2,A3...A7
'
Case 136    'LSB of chip ID
    scratch = VB_Writew(SASEQ_ADDR, chip_cnt & 1)  'write lsb
Case 137
    scratch = VB_Writew(SASEQ_ADDR, (chip_cnt / 2) & 1)  'write next-to-lsb
Case 138
    scratch = VB_Writew(SASEQ_ADDR, (chip_cnt / 4) & 1)  'write bit 2
Case 139, 140   'next two chip id bits are zero
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 141        'then a chip id bit of 1
    scratch = VB_Writew(SASEQ_ADDR, 1)
Case 142        'and one more zero
    scratch = VB_Writew(SASEQ_ADDR, 0)
    bitmask = 16 ' prepare for loading bits 143 thru 147
'
'   strip bits from pipe_depth passed value using bitmask
'   Unfortunately, pipe_depth is loaded MSB to LSB.
'
Case 143, 144, 145, 146, 147
    scratch = VB_Writew(SASEQ_ADDR, (pipe_depth / bitmask) & 1)
    bitmask = bitmask / 2
Case 148, 149   'two of three chip current bits are 0
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 150        'the third bit is a 1
    scratch = VB_Writew(SASEQ_ADDR, 1)
Case 151, 152, 153  'cal voltage bits are all zero
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 154        'as is the adc ramp up bit
    scratch = VB_Writew(SASEQ_ADDR, 0)
'
'   ADC Pedestal is loaded 0,1,0,0
'
Case 155     'the first zero
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 156        'the second bit of the four is a 1
    scratch = VB_Writew(SASEQ_ADDR, 1)
Case 157, 158    'the other two zeroes
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 159        'comparator polarity
    scratch = VB_Writew(SASEQ_ADDR, 0)
'
'   read_all is passed in
'
Case 160
    scratch = VB_Writew(SASEQ_ADDR, CLng(read_all))
Case 161    'read neighbor is always off
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 162    'last chip only set for last chip (duh!)
    If (chip_cnt = 7) Then scratch = VB_Writew(SASEQ_ADDR, 1) Else scratch = VB_Writew(SASEQ_ADDR, 0)
'
'   Ramp trim default is:  0,0,1,0,1,0,1,0,0,0,0
'
Case 163, 164
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 165
    scratch = VB_Writew(SASEQ_ADDR, 1)
Case 166
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 167
    scratch = VB_Writew(SASEQ_ADDR, 1)
Case 168
    scratch = VB_Writew(SASEQ_ADDR, 0)
Case 169
    scratch = VB_Writew(SASEQ_ADDR, 1)
```

```
        Case 170, 171, 172, 173
           scratch = VB_Writew(SASEQ_ADDR, 0)
           bitmask = 128       'this is setup for threshold, cases 174 thru 181
        '
        '   zero_suppress_thresh is passed in, and loaded MSB to LSB.
        '
        Case 174, 175, 176, 177, 178, 179, 180, 181
           scratch = VB_Writew(SASEQ_ADDR, (zero_suppress_thresh / bitmask) & 1)
           bitmask = bitmask / 2
        '
        '   counter modulo default is: 1,0,0,0,0,0,0,1 (129)
        '
        Case 182
           scratch = VB_Writew(SASEQ_ADDR, 1)
        Case 183, 184, 185, 186, 187, 188
           scratch = VB_Writew(SASEQ_ADDR, 0)
        Case 189
        scratch = VB_Writew(SASEQ_ADDR, 1)
     End Select
     Next bit_cnt
Next chip_cnt
'
'   OK, with all 190 bits times 7 chips loaded, clock the shadow register to load it
'   for real.
'
scratch = VB_Writew(SASEQ_BASE_ADDR + &H10, 1)
End Sub
```